Chapter 4. Creating and using bean definitions

4.1. `@Configuration`

Annotating a class with the `@Configuration` annotation indicates that the class will be used by JavaConfig as a source of bean definitions.

An application may make use of just one `@Configuration`-annotated class, or many. `@Configuration` can be considered the equivalent of XML's `<beans/>` element. Like `<beans/>`, it provides an opportunity to explicitly set defaults for all enclosed bean definitions.

```
@Configuration(defaultAutowire = Autowire.BY_TYPE, defaultLazy = Lazy.FALSE)
public class DataSourceConfiguration {
    // bean definitions follow
}
```

Because the semantics of the attributes to the `@Configuration` annotation are 1:1 with the attributes to the `<beans/>` element, this documentation defers to the beans-definition section of Chapter 3, IoC from the Core Spring documentation.

4.2. `@Bean`

`@Bean` is a method-level annotation and a direct analog of the XML `<bean/>` element. The annotation supports most of the attributes offered by `<bean/>` such as init-method, destroy-method, autowiring, lazy-init, dependency-check, depends-on and scope.

4.2.1. Declaring a bean

To declare a bean, simply annotate a method with the `@Bean` annotation. When JavaConfig encounters such a method, it will execute that method and register the return value as a bean within a `BeanFactory`. By default, the bean name will be that of the method name (see bean naming for details on how to customize this behavior).

```
@Configuration
public class AppConfig {
    @Bean
    public TransferService transferService() {
        return new TransferServiceImpl();
    }
}
```

The above is exactly equivalent to the following `appConfig.xml`:

```
<beans>
    <bean name="transferService" class="com.acme.TransferServiceImpl"/>
</beans>
```

Both will result in a bean named `transferService` being available in the `BeanFactory`/`ApplicationContext`, bound to an object instance of type `TransferServiceImpl`:

```
transferService => com.acme.TransferService
```

See Section 4.3, “ JavaConfigApplicationContext ” for details about instantiating and using an `ApplicationContext` with JavaConfig.

4.2.2. Using `*Aware` interfaces

The standard set of `*Aware` interfaces such as BeanFactoryAware, BeanNameAware, MessageSourceAware, ApplicationContextAware, etc. are fully supported. Consider an example class that implements `BeanFactoryAware`:

```
public class AwareBean implements BeanFactoryAware {

    private BeanFactory factory;

    // BeanFactoryAware setter (called by Spring during bean instantiation)
    public void setBeanFactory(BeanFactory beanFactory) throws BeansException {
        this.factory = beanFactory;
    }

    public void close(){
        // do clean-up
    }
}
```

Also, the lifecycle callback methods are fully supported.

4.2.3. Bean visibility

A feature unique to JavaConfig feature is *bean visibility*. JavaConfig uses standard Java method visibility modifiers to determine if the bean ultimately returned from a method can be accessed by an owning application context / bean factory.

Consider the following configuration:

```
@Configuration
public abstract class VisibilityConfiguration {

  @Bean
  public Bean publicBean() {
     Bean bean = new Bean();
     bean.setDependency(hiddenBean());
     return bean;
  }

  @Bean
  protected HiddenBean hiddenBean() {
     return new Bean("protected bean");
  }

  @Bean
  HiddenBean secretBean() {
     Bean bean = new Bean("package-private bean");
     // hidden beans can access beans defined in the 'owning' context
     bean.setDependency(outsideBean());
  }

  @ExternalBean
  public abstract Bean outsideBean()
}
```

Let's bootstrap the above configuration within a traditional XML configuration (for more information on mixing configuration strategies see Chapter 8, *Combining configuration approaches*). The application context being instantiated agaist the XML file will be the 'owning' or 'enclosing' application context, and will not be able to 'see' the hidden beans:

```
<beans>
 <!-- the configuration above -->
 <bean class="my.java.config.VisibilityConfiguration"/>

 <!-- Java Configuration post processor -->
 <bean class="org.springframework.config.java.process.ConfigurationPostProcessor"/>

 <bean id="mainBean" class="my.company.Bean">
    <!-- this will work -->
    <property name="dependency" ref="publicBean"/>
    <!-- this will *not* work -->
    <property name="anotherDependency" ref="hiddenBean"/>
 </bean>
</beans>
```

As JavaConfig encounters the `VisibilityConfiguration` class, it will create 3 beans : `publicBean`, `hiddenBean` and `secretBean`. All of them can see each other however, beans created in the 'owning' application context (the application context that bootstraps JavaConfig) will see only `publicBean`. Both `hiddenBean` and `secretBean` can be accessed only by beans created inside `VisibilityConfiguration`.

Any `@Bean` annotated method, which is not `public` (i.e. with `protected` or default visibility), will create a 'hidden' bean. Note that due to technical limitations, `private @Bean` methods are not supported.

In the example above, `mainBean` has been configured with both `publicBean` and `hiddenBean`. However, since the latter is (as the name imply) hidden, at runtime Spring will throw:

```
org.springframework.beans.factory.NoSuchBeanDefinitionException: No bean named 'hiddenBean' is defined
 ...
```

To provide the visibility functionality, JavaConfig takes advantage of the application context hierarchy provided by the Spring container, placing all hidden beans for a particular configuration class inside a child application context. Thus, the hidden beans can access beans defined in the parent (or owning) context but not the other way around.

4.2.4. Bean scoping

4.2.4.1. Using `@Bean`'s `scope` attribute

JavaConfig makes available each of the four standard scopes specified in Section 3.4, "Bean Scopes" of the Spring reference documentation.

The `DefaultScopes` class provides string constants for each of these four scopes. SINGLETON is the default, and can be overridden by supplying the `scope` attribute to `@Bean` annotation:

```
@Configuration
public class MyConfiguration {
    @Bean(scope=DefaultScopes.PROTOTYPE)
    public Encryptor encryptor() {
        // ...
    }
}
```

4.2.4.2. `@ScopedProxy`

Spring offers a convenient way of working with scoped dependencies through scoped proxies. The easiest way to create such a proxy when using the XML configuration, is the `<aop:scoped-proxy/>` element. JavaConfig offers as alternative the `@ScopedProxy` annotation which provides the same semantics and configuration options.

If we were to port the the XML reference documentation scoped proxy example (see link above) to JavaConfig, it would look like the following:

```
// a HTTP Session-scoped bean exposed as a proxy
@Bean(scope = DefaultScopes.SESSION)
@ScopedProxy
public UserPreferences userPreferences() {
   return new UserPreferences();
}

@Bean
public Service userService() {
   UserService service = new SimpleUserService();
   // a reference to the proxied 'userPreferences' bean
   service.seUserPreferences(userPreferences());
   return service;
}
```

4.2.4.3. Method injection

As noted in the Core documentation, method injection is an advanced feature that should be comparatively rarely used. When using XML configuration, it is required in cases where a singleton-scoped bean has a dependency on a prototype-scoped bean. In JavaConfig, however, it is a (somewhat) simpler proposition:

```
@Bean
public MyAbstractSingleton mySingleton(){
    return new MyAbstractSingleton(myDependencies()){
        public MyPrototype createMyPrototype(){
            return new MyPrototype(someOtherDependency());
            // or alternatively return myPrototype() -- this is some @Bean or @ExternalBean method...
        }
    }
}
```

4.2.5. Bean naming

By default, JavaConfig uses a `@Bean` method's name as the name of the resulting bean. This functionality can be overridden, however, using the `BeanNamingStrategy` extension point.

```
<beans>
    <bean class="org.springframework.config.java.process.ConfigurationPostProcessor">
        <property name="namingStrategy">
            <bean class="my.custom.NamingStrategy"/>
        </property>
    </bean>
</beans>
```

Note

Overriding the bean naming strategy is currently only supported by XML configuration of `ConfigurationPostProcessor`. In future revisions, it will be possible to specify BeanNamingStrategy directly on `JavaConfigApplicationContext`. Watch SJC-86 for details.

For more details, see the API documentation on `BeanNamingStrategy`.

For more information on integrating JavaConfig and XML, see Chapter 8, *Combining configuration approaches*

4.3. `JavaConfigApplicationContext`

`JavaConfigApplicationContext` provides direct access to the beans defined by `@Configuration`-annotated classes. For more information on the ApplicationContext API in general, please refer to the Core Spring documentation.

4.3.1. Construction Options

Instantiating the `JavaConfigApplicationContext` can be done by supplying `@Configuration`-annotated class literals to the constructor, and/or strings representing packages to scan for `@Configuration`-annotated classes.

4.3.1.1. Construction by class literal

Each of the class literals supplied to the constructor will be processed, and for each `@Bean`-annotated method encountered, JavaConfig will create a bean definition and ultimately instantiate and initialize the bean.

```
JavaConfigApplicationContext context =
    new JavaConfigApplicationContext(AppConfig.class);
Service service = context.getBean(Service.class);
```

```
JavaConfigApplicationContext context =
    new JavaConfigApplicationContext(AppConfig.class, DataConfig.class);
Service service = context.getBean(Service.class);
```

4.3.1.2. Construction by base package

Base packages will be scanned for the existence of any `@Configuration`-annotated classes. Any candidate classes will then be processed much as if they had been supplied directly as class literals to the constructor.

```
JavaConfigApplicationContext context =
    new JavaConfigApplicationContext("**/configuration/**/*.class");
Service service = (Service) context.getBean("serviceA");
```

```
JavaConfigApplicationContext context =
    new JavaConfigApplicationContext("**/configuration/**/*.class", "**/other/*Config.class);
Service service = (Service) context.getBean("serviceA");
```

4.3.1.3. Post-construction configuration

When one or more classes/packages are used during construction, a `JavaConfigApplicationContext` cannot be further configured. If post-construction configuration is preferred or required, use either the no-arg constructor, configure by calling setters, then manually refresh the context. After the call to `refresh()`, the context will be 'closed for configuration'.

```
JavaConfigApplicationContext context = new JavaConfigApplicationContext();
context.setParent(otherConfig);
context.setConfigClasses(AppConfig.class, DataConfig.class);
context.setBasePackages("**/configuration/**/*.class");
context.refresh();
Service service = (Service) context.getBean("serviceA");
```

Note

Whenever multiple packages and/or classes are used to instantiate a `JavaConfigApplicationContext`, *order matters*. This is important when considering what happens if two configuration classes define a bean with the same name. The last-specified class wins.

4.3.2. Accessing beans with `getBean()`

`JavaConfigApplicationContext` provides several variants of the `getBean()` method for accessing beans.

4.3.2.1. Type-safe access

The preferred method for accessing beans is with the type-safe `getBean()` method.

```
JavaConfigApplicationContext context = new JavaConfigApplicationContext(...);
Service service = context.getBean(Service.class);
```

4.3.2.1.1. Disambuguation options

If more than one bean of type `Service` had been defined in the example above, the call to `getBean()` would have thrown an exception indicating an ambiguity that the container could not resolve. In these cases, the user has a number of options for disambiguation:

4.3.2.1.1.1. Indicating a `@Bean` as `primary`

Like Spring's XML configuration, JavaConfig allows for specifying a given `@Bean` as `primary`:

```
@Configuration
public class MyConfig {
    @Bean(primary=Primary.TRUE)
    public Service myService() {
        return new Service();
    }

    @Bean
    public Service backupService() {
        return new Service();
    }
}
```

After this modification, all calls to `getBean(Service.class)` will return the `primary` bean

```
JavaConfigApplicationContext context = new JavaConfigApplicationContext(...);
// returns the myService() primary bean
Service service = context.getBean(Service.class);
```

4.3.2.1.1.2. Disambiguation by bean name

JavaConfig provides a `getBean()` variant that accepts both a class and a bean name for cases just such as these.

```
JavaConfigApplicationContext context = new JavaConfigApplicationContext(...);
Service service = context.getBean(Service.class, "myService");
```

Because bean ids must be unique, this call guarantees that the ambiguity cannot occur.

4.3.2.1.1.3. Retrieve all beans of a given type

It is also reasonable to call the `getBeansOfType()` method in order to return all beans that implement a given interface:

```
JavaConfigApplicationContext context = new JavaConfigApplicationContext(...);
Map matchingBeans = context.getBeansOfType(Service.class);
```

Note that this latter approach is actually a feature of the Core Spring Framework's `AbstractApplicationContext` (which `JavaConfigApplicationContext` extends) and is not type-safe, in that the returned `Map` is not parameterized.

4.3.2.2. String-based access

Beans may be accessed via the traditional string-based `getBean()` API as well. Of course this is not type-safe and requires casting, but avoids any potential ambiguity entirely:

```
JavaConfigApplicationContext context = new JavaConfigApplicationContext(...);
Service service = (Service) context.getBean("myService");
```

Prev Home Next

Part II. API Reference Sponsored by SpringSource Chapter 5. Modularizing configurations